RDF Object Type and Reification in Oracle

Nicole Alexander Siva Ravada

Oracle Corporation
1 Oracle Drive
Nashua, NH 03062
USA
{Nicole.Alexander, Siva.Ravada}@oracle.com

Abstract

Resource Description Framework (RDF) is a standard for representing information that can be identified using a Universal Resource Identifier. In particular, it is intended for representing metadata about Web resources. RDF is being used in numerous application areas, including Life Sciences, Digital Libraries, and Intelligence. The RDF data structure is a directed graph or network. Current solutions to managing RDF data utilize flat relational tables for database storage. This paper presents an alternative approach to managing RDF data in the database. We introduce a new Oracle object type for storing RDF data. The object type is built on top of the Oracle Spatial Network Data Model (NDM), which is Oracle's network solution in the database. This exposes the NDM functionality to RDF data, allowing RDF data to be managed as objects and analysed as networks. Reification, a means of providing metadata for the RDF data, puts a strain on storage. We present a streamlined approach to representing reified RDF data for faster retrievals. An RDF object type and reification in Oracle provide the basic infrastructure for effective metadata management.

1. Introduction

Resource Description Framework (RDF) is a standard for representing information that can be identified using a Universal Resource Identifier (URI). In particular, it is intended for representing metadata about Web resources [1]. RDF is being used to represent data in numerous application areas, including Life Sciences, Digital Libraries, Intelligence, E-Commerce, and Personal Information Management. RDF provides the basic building blocks for supporting the Semantic Web [2]. The simplicity of its structure promotes interoperability across applications, and its machine-understandable format facilitates the automated processing of Web resources [3].

To represent data in RDF, each statement is broken into a <subject, predicate, object> triple. This triple is effectively modelled as a directed graph [4]: The *subject* and *object* of the triple are modelled as nodes, and the *predicate* (or property) as a directed link that describes the relationship between the nodes. The direction of the link always points towards the object (Figure 1).

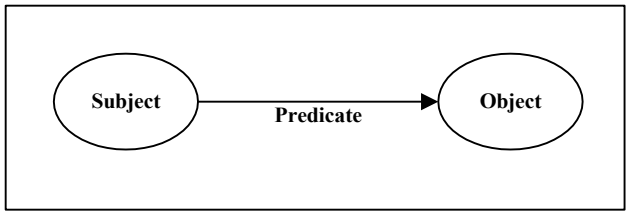

Figure 1: RDF Triple in Directed Graph Data Model

A number of RDF storage systems and browsers are available [5]: *Jena2* [6], *Kowari* [7], *Sesame* [8], and *Longwell* [9], to name a few. Many of the available systems implement persistent storage using relational databases, where data is stored in flat relational tables: triples are stored in a main statement table with links to supporting tables; this statement table has columns for the subject, predicate, and object; and each row represents a triple [10] [6].

In this paper we present an alternative approach to managing RDF data. We introduce a new Oracle object type (SDO_RDF_TRIPLE_S) for storing RDF data [11]. There are many advantages to managing RDF data as a database object [12]:

- It makes it easier to model RDF applications.
- RDF data can be easily integrated with other enterprise data.
- Reusability of the RDF object makes it possible to develop applications more efficiently.

- Object abstraction and the encapsulation of RDF-specific behaviours make applications easier to understand and maintain.
- No mapping is required between client-side RDF objects, and database columns and tables that contain triples.
- No additional configurations are required for storing RDF data.

The RDF object type is built on top of the Oracle Spatial Network Data Model (NDM) [13]. NDM is Oracle's optimal solution for storing, managing, and analysing networks or graphs in the database. The RDF directed graph structure is therefore effectively modelled in NDM as a directed logical network. All the NDM functionality is exposed to RDF data. Additionally, RDF-specific functionality has been added with the new object type.

RDF data is stored in a central schema in Oracle: there is one universe for all RDF data in the database, and user-level access functions and constructors are provided for query and update. Triples are parsed and stored in the system as entries in the NDM node$ and link$ tables. Nodes in the RDF network are uniquely stored and are reused when encountered in incoming triples. In user-defined application tables, only references are stored in the SDO_RDF_TRIPLE_S object to point to the actual triple stored in the central schema. This central storage of RDF data allows analysis to be readily performed across all applications in the database or on selected applications.

Consider the following hypothetical scenario of RDF data storage by different members of the Intelligence Community (IC). The Central Intelligence Agency (CIA), Department of Homeland Security (DHS), and the Federal Bureau of Investigation (FBI) each has its separate RDF-based application stored in Oracle. Figure 2 shows the RDF data stored by the CIA, DHS, and FBI, respectively. Each member of the IC has entered the triple: <gov:files, gov:terrorSuspect, id:JohnDoe>. In their application, each member of the IC will have the same subject ID, predicate ID, and object ID for the repeated triple entered.

The RDF data for these three applications is stored in a central schema. Therefore, reasoning can be done over the entire database to help combat terrorism, though each member of the IC manages its own RDF data separately.

CIA Data
<gov:files, gov:terrorSuspect, id:JohnDoe>
<gov:files, gov:terrorSuspect, id:JaneDoe>

DHS Data
<id:JohnDoe, gov:terrorAction, Bombing>
<gov:files, gov:terrorSuspect, id:JohnDoe>

FBI Data
<id:JohnDoe, gov:enteredCountry, June-20-2000>
<gov:files, gov:terrorSuspect, id:JohnDoe>

Figure 2: RDF Data for IC Applications

In some applications, there is the need to provide metadata for the RDF triples, for example, who created them and the date of their creation. In RDF this is done using *reification*. *Reification* is the description of an RDF triple using the RDF built-in vocabulary [1]. When implemented naïvely, *reification,* described as "The RDF Big Ugly" [14], significantly bloats storage and inflates query times, since four new triples are stored for each reification. We tame the "Big Ugly" by utilizing an Oracle XML DB DBUri to directly reference the reified triple in the database: only one new triple is stored for each reification.

Initial results show comparable performance between queries using the RDF object type and a relational-based storage implementation. However, further optimisations for the RDF object type are expected in the future. The contributions of this paper are therefore a more convenient way to manage RDF applications within the database, and a direct approach to reifying RDF data with less performance overhead.

Section 2 reviews RDF concepts. Section 3 reviews related work. Section 4 outlines RDF data storage in Oracle. Section 5 describes how reification is implemented in Oracle. Section 6 reviews query and analysis of RDF data. Section 7 reviews performance, and section 8 concludes and outlines future work.

2. RDF Concepts

A complete description of the RDF language, including its concepts, abstract syntax, and semantics, is available on the Website: http://www.w3.org/RDF/. In this section we briefly review the RDF concepts (*triples*, *graphs*, *containers,* and *reification*) mentioned in this paper.

Each RDF statement describing a resource is represented using a *triple*. And a set of triples is known as an RDF *graph* or *model*. The components of a triple are the *subject*, represented by a *URI* or a *blank node*; the *predicate* or *property*, represented by a *URI*; and the *object*, represented by a *URI*, a *blank node*, or a *literal* [4] [1]:

- A *URI* (e.g. http://www.w3.org/1999/02/22-rdf-syntax-ns#subject) is a more general form of Uniform Resource Locator (URL). It allows information about a resource to be recorded without the use of a specific network address.
- A *blank node* is used when a subject or object node is unknown. It is also used when the relationship between a subject and an object node is n-ary (as is the case with *RDF* containers). A blank node is prefixed with the identifier "_:" and has the form _:anyname001.
- A *literal* is basically a sting with an optional language tag. It is used to represent values like names, dates, and numbers.
- A *typed literal* (e.g."25"^^http://www.w3.org-/2001/XMLSchema#int) is a string combined with a datatype. The datatype is always a URI.

To describe groups of things in RDF (for example, to illustrate that a class has several students), a resource called a *container* is used. The participants of a container are called *members*. To represent a container and its members, a blank node is typically generated for the container, and each member is attached to this node as the object of a triple.

Reification is the description of a triple using the RDF vocabulary. The reification vocabulary consists of four statements, which make up the reification quad. For example, the triple <S,P,O> reified by a resource R is represented by the four triples:

- <R, rdf:type, rdf:Statement>
- <R, rdf:subject, S>
- <R, rdf:property, P>
- <R, rdf:object, O>

Assertions can then be made about triple <S,P,O> , using R, for example, <N, ora:said, R>. Reification therefore allows triples to be attached as properties to other triples [14].

In following sections we describe how these concepts are supported in Oracle and other systems.

3. Related Work

A number of RDF storage systems have implemented persistent storage using relational databases. Beckett [10] reports on several of these systems, describing their schemas. Responses to Melnik [15], also yield more schema designs for RDF storage in relational databases. The overall design approach has been to store triples in a main statement table, with links to supporting tables. The main statement table has columns for the subject, predicate, and object, and each row represents a triple.

In this section we review the Jena2 schema design, because it has departed somewhat from the typical design approach to RDF storage in relational databases.

3.1 Jena2

Jena is a leading Semantic Web programmers' toolkit [16], currently in its second generation of the product [6]. Jena2 is designed to overcome some of the problems of Jena1. Specifically, the altered schema design is to address performance issues due to too many table joins, and storage bloat resulting from a naïve implementation of reification [6].

Jena1 utilized a normalized triple store approach [17]. A statement table stored references to the subject, predicate, and object, and the actual text values for the URIs and the literals were stored in two additional tables. This design was efficient on space, because text values were only stored once, regardless of the number of times they occurred in triples. However, a three-way join was required for "find" operations [6]. Also, the single statement table did not scale for large datasets [6].

Jena2 utilizes a denormalized [18] [6], multi-model triple store approach [17]. Models are stored in separate tables, and each model stores asserted statements in one table and reified statements in another [18] – although these default behaviours can be altered through configuration. The asserted statement table stores the actual text values for the triples in subject, predicate, object columns. Text values are therefore stored redundantly, whenever the same text value appears in different triples. Jena2 thereby consumes more storage space than Jena1; however, the number of required table joins are reduced at query time, improving performance. Reified statements are stored in a property-class table [19] [6] that has columns StmtURI (representing the reified statement), rdf:subject, rdf:predicate, rdf:object, and rdf:type. A single row with all attributes present represents a reified triple.

In addition to the tables for asserted and reified statements, Jena2 can be configured to include property tables on graph creation [19] [6]. These tables store subject-value pairs for specified predicates. The table columns include the subject, and each predicate to be represented in the table. An example Dublin Core [20] property table may include the columns subject, dc:title, dc:publisher, and dc:description [19], where a single row stores the predicate values for a common subject. Property tables are said to provide modest storage reduction, since predicate URIs are not stored [6]. They attempt to cluster properties that are commonly accessed together and thereby improve performance.

These are the basic tables that comprise the Jena2 schema; other supporting tables are described in Jena [17]. Jena2, as well as the other systems that implement persistent storage in relational databases, utilize flat relational tables. In the next section we describe the schema used by Oracle, and how RDF objects are implemented in the database.

4. Data Storage in Oracle

The RDF store in Oracle is built on top of the Oracle Spatial Network Data Model (NDM), in which a network or graph captures relationships between objects using connectivity. NDM supports both directed and undirected spatial and logical networks. It has two components: a database schema and a Java API [13].

RDF graphs are modelled as a directed logical network in NDM. In this network, the subjects and objects of triples are mapped to nodes, and predicates *(or* properties*)* are mapped to links that have subject start-nodes and object end-nodes. A link, therefore, represents a complete RDF triple. A key feature of RDF storage in Oracle is that nodes are stored only once – regardless of the number of times they participate in triples. But a new link is always created whenever a new triple is inserted. Figure 3 illustrates how an RDF graph of three RDF triples: <S1, P1, O1>; <S1, P2, O2>; <S2, P2, O2> is modelled. When a triple is deleted from the database, the corresponding link is removed. However, the nodes attached to this link are not removed if there are other links connected to them.

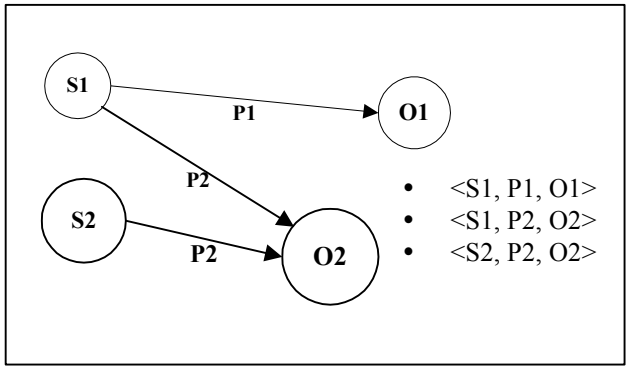

Figure 3: RDF Graph in Oracle

In Oracle, RDF triples are parsed and stored in global tables under a central schema. All the triples for all the RDF graphs (or models) in the database are stored under this schema. Only references (IDs) to these triples are stored in the user-defined application tables that may contain other attributes related to the triples. And functions are provided to retrieve the actual triples when necessary. The required tables for RDF data storage are: rdf_model$, rdf_value$, rdf_node$, rdf_link$, and rdf_blank_node$. The rdf_node$ and rdf_link$ tables are required by NDM: they expose the NDM networking capabilities to RDF data.

The rdf_model$ table stores the names and IDs of all the RDF graphs in the database. It has the following columns:

- MODEL_ID: the unique identifier and primary key for this table.
- MODEL_NAME: the name of the RDF graph.

The rdf_value$ table stores the text values (i.e. URIs, blank nodes, and literals) for a triple. Each text entry is uniquely stored. This table has the following columns:

- VALUE_ID: the unique identifier and primary key for this table.
- VALUE_NAME: the actual text value for the URI, blank node, or literal. It is stored as an XML DB URIType [21].
- VALUE_TYPE: the type of text value stored in the VALUE_NAME column. URI (UR), blank node (BN), plain literal (PL), typed literal (TL), plain long-literal (PLL), and typed long-literal (TLL) are some of the supported types.
- LITERAL_TYPE: the datatype URI for a typed literal or typed long-literal.
- LONG_VALUE: the plain long-literal or typed long-literal text value; long-literals are text values that exceed 4000 characters.

The rdf_node$ table stores the IDs of text values that participate as subjects or objects of triples. Each ID is uniquely stored in this table – regardless of the number of times a node is used in triples. This table has the following columns:

- NODE_ID: the unique identifier and primary key for this table.
- ACTIVE: the status of a node in the network (Y or N).

The rdf_link$ table stores the triples for all the RDF graphs in the database. It is partitioned by graphs for improved query performance. This table includes the following columns:

- LINK_ID: the unique triple identifier and part of the primary key.
- START_NODE_ID: the VALUE_ID for the text value of the subject of the triple. Also part of the primary key.
- P_VALUE_ID: the VALUE_ID for the text value of the predicate of the triple.
- END_NODE_ID: the VALUE_ID for the text value of the object of the triple. Also part of the primary key.
- LINK_TYPE: the type of predicate represented by the URI of the P_VALUE_ID. STANDARD, RDF_MEMBER, RDF_TYPE, and RDF_* are some of the supported types.
- ACTIVE: the status of a link in the network (Y or N).
- CONTEXT: tells whether a triple was directly entered into the database or inferred as part of a reification statement (D or I).
- REIF_LINK: tells whether the START_NODE_ID, P_VALUE_ID, or END_NODE_ID references a reified triple (Y or N).

- MODEL_ID: the ID for the RDF graph to which the triple belongs. It logically partitions the table by RDF graphs.

The rdf_blank_node$ table optionally stores the names and IDs of blank nodes that are to be reused when encountered in incoming triples. Blank-nodes' values are generally not reused like other text values. However, for entering containers, and other situations were a blank node's value is common to several triples, information can be stored in the rdf_blank_node$ table, so that the structure of the incoming triples is correctly modelled in the database. This table has the following columns:

- NODE_ID: the VALUE_ID for the blank node currently in the database.
- NODE_NAME: the globally unique, system-generated blank node text value. It has the form _:ORABN*uniqueID*.
- ORIG_NAME: the name given to the blank node by an external application or other Oracle RDF data store, before it was entered into the database.
- MODEL_ID: the ID for the RDF graph from which the blank node can be reused.

Figure 4 summarizes the RDF tables and their connections.

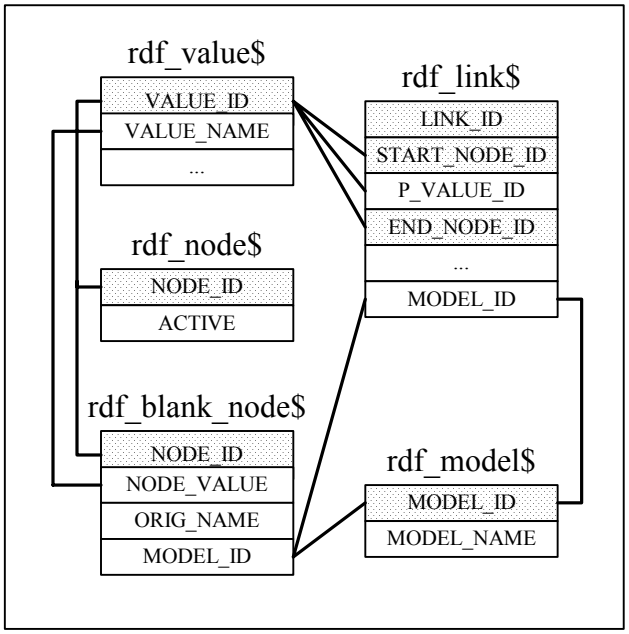

Figure 4: RDF Tables in Oracle

4.1 Parsing Triples

When a user attempts to insert a triple, the rdf_model$ table is first checked to ensure that the RDF graph exists. The rdf_value$ table is then checked to determine if the text values for each part of the triple (subject, predicate, object) already exist in the database. If they already exist, their VALUE_IDs are retrieved. The rdf_link$ table is then checked to determine if the triple already exists in the specified graph. If the triple already exists in the specified graph, the IDs for the previously inserted triple are returned and no new inserts are made.

If the text values for the triple are not found in the rdf_value$ table, they are inserted into the table and assigned new VALUE_IDs. The VALUE_IDs corresponding to the subject and the object of the triple are inserted into the rdf_node$ table. In the rdf_link$ table, a new LINK_ID is then generated for the triple. The VALUE_ID for the predicate of the triple becomes the P_VALUE_ID in the rdf_link$ table, and the VALUE_IDs for the subject and object of the triple become the START_NODE_ID and END_NODE_ID, respectively. A new record is inserted into the rdf_link$ table whenever a new triple is entered into a graph.

Inserted triples may contain blank nodes as their subject or object. When encountered, blank nodes are automatically renamed to globally unique, Oracle system-generated blank nodes of the form _:ORABN*uniqueID*. By default, blank nodes are not reused by the system. However, a user has the option to specify whether a particular blank node is to be reused by incoming triples. In such a case, the blank node's original name, along with the Oracle system-generated name and VALUE_ID are stored in the rdf_blank_node$ table. Future incoming blank nodes with the same original name and belonging to the same RDF graph will cause the existing blank node in the database to be reused. To load containers, blank nodes must be reused. Entries in the rdf_blank_node$ table can be safely deleted when their reuse is no longer required.

4.2 RDF Object Types

Two new object types have been defined to manage RDF data in Oracle: SDO_RDF_TRIPLE and SDO_RDF-_TRIPLE_S. The SDO_RDF_TRIPLE object is defined to serve as the triple representation of RDF data <subject, predicate, object> (Figure 5). The SDO_RDF_TRIPLE_S (RDF triple storage) object is defined to store persistent RDF data in the database. The RDF triple storage object contains only IDs (Figure 6) that point to the triple maintained in the central schema. Several constructors and member functions are provided with the storage object.

Figure 5 shows generalized code extracts for the RDF objects. The SDO_RDF_TRIPLE type provides a triple view of the data, and the SDO_RDF_TRIPLE_S types stores the IDs for the triple. The IDs stored are:

- rdf_t_id: the unique triple ID, which is the LINK_ID in the rdf_link$ table.
- rdf_m_id: the MODEL_ID for the graph, which is the MODEL_ID in the rdf_link$ table.
- rdf_s_id: the VALUE_ID for the subject of the triple, which is the START_NODE_ID in the rdf_link$ table.

- rdf_p_id: the VALUE_ID for the predicate of the triple, which is the P_VALUE_ID in the rdf_link$ table.
- rdf_o_id: the VALUE_ID for the object of the triple, which is the END_NODE_ID in the rdf_link$ table.

```
CREATE TYPE sdo_rdf_triple AS OBJECT (
    subject VARCHAR2(4000),
    property VARCHAR2(4000),
    object VARCHAR2(10000)
) ;

CREATE TYPE sdo_rdf_triple_s AS OBJECT (
    rdf_t_id NUMBER,
    rdf_m_id NUMBER,
    rdf_s_id NUMBER,
    rdf_p_id NUMBER,
    rdf_o_id NUMBER,
    CONSTRUCTOR FUNCTION sdo_rdf_triple_s (
            model_name VARCHAR2,
            subject VARCHAR2,
            property VARCHAR2,
            object VARCHAR2
    ) RETURN SELF,

MEMBER FUNCTION get_triple
        RETURN SDO_RDF_TRIPLE,
MEMBER FUNCTION get_subject
        RETURN VARCHAR2,
MEMBER FUNCTION get_property
        RETURN VARCHAR2,
MEMBER FUNCTION get_object
        RETURN CLOB,
...
```

Figure 5: RDF Objects and Member Functions

The member function GET_TRIPLE() returns the SDO_RDF_TRIPLE type with the subject, predicate, and object of the triple. The member functions GET_SUBJECT(), GET_PROPERTY() and GET_OBJECT() return the subject, predicate, and object, respectively. The GET_OBJECT() function returns a CLOB type, since the returned object may be a long literal.

Figure 6 illustrates how the RDF data for the IC in Figure 2 is stored in the application tables. The repeated triple <gov:files, gov:terrorSuspect, id:JohnDoe> shares the same RDF_S_ID, RDF_P_ID, and RDF_O_ID.

CIA_TRIPLE (RDF_T_ID, RDF_M_ID, RDF_S_ID, RDF_P_ID, RDF_O_ID)
SDO_RDF_TRIPLE_S (2051, 7, 1068, 1070, 1069)
SDO_RDF_TRIPLE_S (2052, 7, 1068, 1070, 1071)

DHS_TRIPLE (RDF_T_ID, RDF_M_ID, RDF_S_ID, RDF_P_ID, RDF_O_ID)
SDO_RDF_TRIPLE_S (2053, 8, 1069, 1073, 1072)
SDO_RDF_TRIPLE_S (2054, 8, 1068, 1070, 1069)

FBI_TRIPLE (RDF_T_ID, RDF_M_ID, RDF_S_ID, RDF_P_ID, RDF_O_ID)
SDO_RDF_TRIPLE_S (2055, 9, 1069, 1075, 1074)
SDO_RDF_TRIPLE_S (2056, 9, 1068, 1070, 1069)

Figure 6: The SDO_RDF_TRIPLE_S Object

4.3 Creating Graphs and Inserting Triples

The SDO_RDF PL/SQL package provides functions and procedures for managing the SDO_RDF_TRIPLE_S object. It includes the function for creating graphs. To build an RDF application, the following steps are performed:

1. Create a Graph
   - EXECUTE SDO_RDF.CREATE_RDF_MODEL('cia');
2. Create an Application Table with the RDF Object
   - CREATE TABLE ciadata (id NUMBER, cia_triple SDO_RDF_TRIPLE_S);
3. Insert Triples into the Application Table
   - INSERT INTO ciadata VALUES (1, SDO_RDF_TRIPLE_S('cia' , 'gov:files', 'gov:terrorSuspect' , 'id:JohnDoe'));

5. Implementing Reification in Oracle

To represent a reified triple, a resource is generated using the triple's unique RDF_T_ID (or LINK_ID). An Oracle XML DB DBUri that points directly to the triple's RDF_T_ID is used as the resource. A DBUri is a URI that points to a set of rows, a single row, or a single column in a database [21]. The DBUriType object represents a direct link to data in a table, and object methods can be called to retrieve this data.

To reify the triple RDF_T_ID: 2051 (in Figure 6), the DBUriType resource generated is /ORADB/MDSYS/RDF_LINK$/ROW[LINK_ID=2051]. A triple </ORADB/MDSYS/RDF_LINK$/ROW[LINK_ID=2051], rdf:type, rdf:Statement> (Figure 7) is entered into the database and parsed in a similar manner to other triples, except its REIF_LINK is set to Y. The rdf:type is the only portion of the reification quad that is stored by Oracle. To determine if a triple is reified in a specified graph, a search is done for its DBUriType.

A Java API is provided for reading reification quads and converting them into reified statements in Oracle. The

Java API calls the SDO_RDF_TRIPLE_S constructor functions for reifying triples and making assertions. Three of the constructors are reviewed in this section:

- **SDO_RDF_TRIPLE_S (model_name, rdf_t_id)**: the reification constructor, which generates a triple of the form </ORADB/MDSYS/RDF_LINK$/ROW[LINK_ID=rdf_t_id], rdf:type, rdf:Statement> in the database.
- **SDO_RDF_TRIPLE_S (model_name, subject, property, rdf_t_id)**: an assertion constructor, which calls the reification constructor (if the triple was not previously reified) and makes an assertion about the triple identified by the rdf_t_id.
- **SDO_RDF_TRIPLE_S (model_name, reif_sub, reif_prop, subject, property, object)**: an assertion constructor, which allows reification of an implied statement – a statement that is not currently in the database. It first inserts the base triple (subject, property, object) and then calls the reification constructor before making the assertion. This assertion constructor can also be used to make an assertion when the base triple is itself an assertion (that is, it exists in the database).

5.1 Reifying and Asserting Direct Triples

Triples entered into the database (as shown in previous examples) are considered assertions – facts, and given the CONTEXT attribute of 'D' for *direct*. A direct triple is one that exists in the database before an attempt is made to reify it.

To reify the triple <gov:files, gov:terrorSuspect, id:JohnDoe>, which has RDF_T_ID: 2051, the following insert is made:

- INSERT INTO ciadata VALUES (3, SDO_RDF_TRIPLE_S('cia', 2051));

To make an assertion about a direct statement, for example, that MI5 said <gov:files, gov:terrorSuspect, id:JohnDoe> (Figure 7), the following insert is made:

- INSERT INTO ciadata VALUES (4, SDO_RDF_TRIPLE_S('cia', 'gov:MI5', 'gov:source', 2051));

5.2 Reifying and Asserting Indirect Triples

Triples entered into the database as the base triple of reification statements only are considered indirect statements and given the CONTEXT attribute of 'I'. For example, in the statement "Interpol said that JohnDoeJr is a terrorSuspect", the triple <gov:files, gov:terrorSuspect, id:JohnDoeJr> is not considered an assertion, since it did not previously exist as a triple before an attempt was made to reify it. It is considered an implied statement, and during reasoning over the database it will be evaluated based on the CIA's *trust* in Interpol. The following insert is made for asserting the implied statement:

- INSERT INTO ciadata VALUES (5, SDO_RDF_TRIPLE_S('cia', 'gov:Interpol', 'gov:source', 'gov:files', 'gov:terrorSuspect' , 'id:JohnDoeJr'));

Note: If the triple <gov:files, gov:terrorSuspect, id:JohnDoeJr> is subsequently entered into the database as an assertion, the CONTEXT for this triple is changed from 'I' to 'D'.

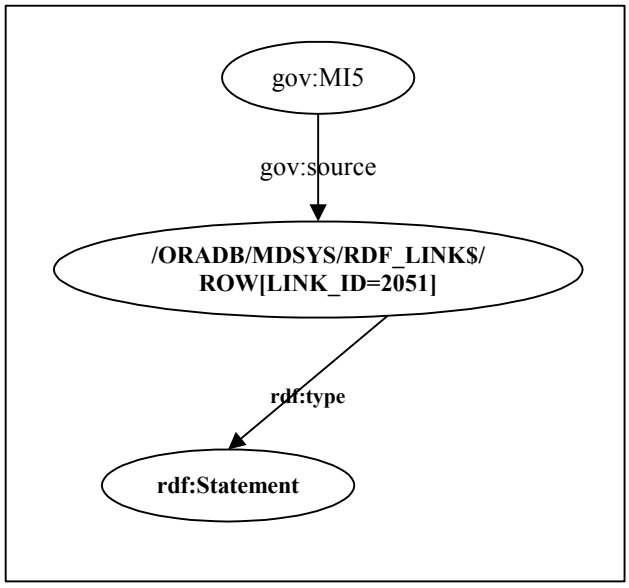

Figure 7: Reified Statement and Assertion

6. Query and Analysis of RDF Data

SQL-level access to RDF data is available through the RDF object's member functions and the SDO_RDF PL/SQL package. The member functions include GET_TRIPLE(), GET_SUBJECT(), GET_PROPERTY(), and GET_OBJECT(). Figure 8 illustrates the use of the GET_TRIPLE() member function. The SDO_RDF package includes functions for querying and accessing the RDF data, for example, IS_TRIPLE(), IS_REIFIED(), GET_TRIPLE_ID(), etc. The data can also be queried using the NDM SDO_NET package for accessing and managing networks in the database.

A Java API is available to perform analysis on RDF data, for example, to find the properties between two resources, or to find the properties between two resources, with links of a specified type. SQL functions for inference-type operations on RDF data are currently in development.

```
SQL> SELECT c.cia_triple.GET_TRIPLE() triple
FROM ciadata c;

TRIPLE (SUBJECT, PROPERTY, OBJECT)
----------------------------------------------------------------
SDO_RDF_TRIPLE
('gov:files', 'gov:terrorSuspect', 'id:JohnDoe')
SDO_RDF_TRIPLE
('gov:files', 'gov:terrorSuspect', 'id:JaneDoe')
SDO_RDF_TRIPLE
('/ORADB/MDSYS/RDF_LINK$ROW[LINK_ID=2051]',
'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
'http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement')
SDO_RDF_TRIPLE
('gov:MI5', 'gov:source',
'/ORADB/MDSYS/RDF_LINK$/ROW[LINK_ID=2051]')
SDO_RDF_TRIPLE
('gov:Interpol', 'gov:source',
'/ORADB/MDSYS/RDF_LINK$/ROW[LINK_ID=2059]')
```

Figure 8: The GET_TRIPLE() Member Function

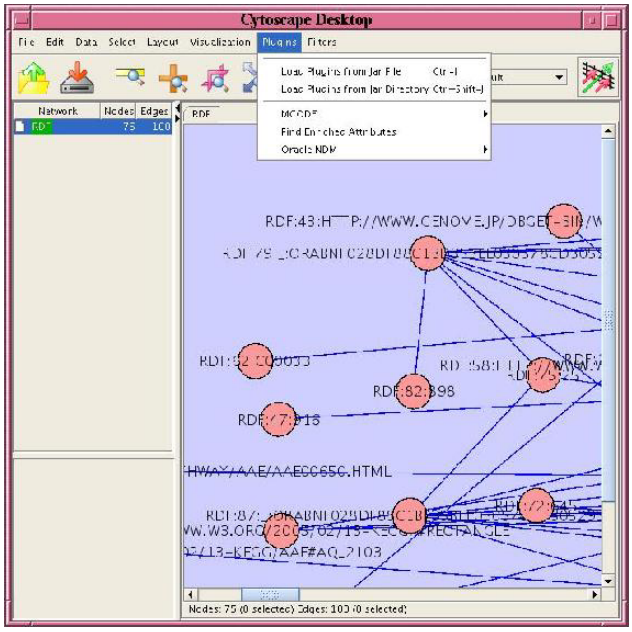

Figure 9: KEGG RDF Network in Cytoscape

6.1 Visualization and Integration in Cytoscape

Cytoscape is an open-source bioinfomatics software platform. It is used for visualizing and integrating biomolecular interaction networks with gene expression profiles and other state data [22] [23]. Cytoscape has an extensible plug-in architecture, which allows additional features and computation analyses to be supported by the platform [22]. NDM provides a plug-in for Cytoscape 2.0, which enables a read/write interface between the programs, as well as the addition of the NDM network-analysis functionality in Cytoscape. Using this plug-in, biomolecular RDF networks in Oracle can be directly accessed in Cytoscape. Figure 9 illustrates an extract of KEGG [24] [25] data as an Oracle RDF network in Cytoscape.

7. Performance

Our initial performance goal for the RDF objects was to ensure that their convenience was not mitigated by slow performance times. To test this, we loaded a UniProt [26] dataset of 5 million triples into Oracle's RDF storage objects. We then checked the query times to retrieve data using the object's member functions, against the times to retrieve data by directly querying the storage tables. In all tested cases, the member functions performed either similarly or slightly better. To attain these performance times for the RDF objects, indexes, and especially function-based indexes, are required on the application table.

7.1 Indexing for Good Performance

When possible, indexes should be built on conditions in the WHERE clause. The following examples show the suggested indexes for three queries on the sample dataset:

- CREATE INDEX up5m_tid_idx ON uniprot5m (triple.rdf_t_id);
  Q1: SELECT u.triple.GET_TRIPLE() FROM uniprot5m u WHERE u.triple.rdf_t_id=4000000;
- CREATE INDEX up5m_sub_fbidx ON uniprot5m (triple.GET_SUBJECT());
  Q2: SELECT u.triple.GET_TRIPLE() FROM uniprot5m u WHERE u.triple.GET_SUBJECT()= 'urn:lsid:uniprot.org:uniprot:P42645';
- CREATE INDEX up5m_obj_fbidx ON uniprot5m (TO_CHAR(triple.GET_OBJECT()));
  Q3: SELECT u.triple.GET_TRIPLE() FROM uniprot5m u WHERE TO_CHAR(u.triple.GET_OBJECT())= 'urn:lsid:uniprot.org:uniprot:P15711';

Table 1 shows the result times for these three queries on the 5 million triple UniProt dataset, using the above indexes. The test machine for the queries is an Intel® Xeon ® 2 CPU 2.8 GHz processors, with Red Hat Linux 3.2.3-26 operating system and Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production (SGA size 468 MB) installed.

Query	Time (sec)	Rows
Q1	0.01	1
Q2	0.04	28
Q3	0.01	4

Table 1: Query Times on the 5M-UniProt Dataset

7.2 Reification

Reification in Oracle requires only 25% of the storage required by naïve implementations, which store the entire reification quad. Queries, for example, IS_REIFIED(), are therefore based on a single row retrieval. Reified triples are parsed and stored similarly to other triples in the database; therefore, no additional overhead is required to manage them. Also, operations on non-reified triples work on reified triples.

There is, however, an initial set-up cost for reification of large datasets: In our current implementation, large datasets require increased Java Virtual Memory heap size, and temporary tablespace for loading. This is due to the fact that the entire input file must be read before inserting triples into the database. The temporary tables are deleted at the end of the loading process.

8. Conclusion and Future Work

An RDF object type and reification in Oracle provide the basic infrastructure for effectively managing RDF data in the database. Storing RDF data as a database object allows RDF applications to be quickly developed and easily maintained. It also allows RDF data to be readily integrated and managed with other enterprise data.

The RDF object type is built on top of NDM, allowing RDF triples to be managed as objects and analyzed as networks. The NDM functionality, as well as third-party applications that support NDM, can be utilized by RDF data.

A streamlined approach to reification utilizes less storage and reduces performance overhead for querying. The URI used in its implementation provides a direct link to the reified triple.

A Java API with a complete suite of functions for managing and analysing RDF data is currently in development. Also, SQL-level reasoning functions that permit user-defined rules bases are being developed. We continue to explore innovative ways to accelerate data retrieval, and we continue our data-management effort to support the Semantic Web.

9. Acknowledgements

Cheng-Hua Wang is the lead NDM developer, we are especially grateful for his contributions. We are also thankful to George Eadon of Oracle Index Organized Tables, Susie Stephens, the Life Sciences Product Manager, and to Melliyal Annamalai of Oracle interMedia. Mike Horhammer, Dan Abugov, and Ning An of Oracle Spatial have given generously of their time, and Chuck Murray has carefully reviewed this manuscript.

10. References

[1] W3C, *RDF Primer*, http://www.w3.org/TR/rdf-primer/, (as of 02/24/2005).

[2] W3C, *Semantic Web,* http://www.w3.org/2001/sw/, (as of 02/24/2005).

[3] W3C, *Frequently Asked Questions about RDF*, http://www.w3.org/RDF/FAQ2003, (as of 02/24/2005).

[4] W3C, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, http://www.w3.org/TR/rdf-concepts/, (as of 02/24/2005).

[5] D. Beckett, *SWAD-Europe Deliverable 10.1: Scalability and Storage: Survey of Free Software / Open Source RDF storage systems*, http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/, (as of 02/24/2005).

[6] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, *Efficient RDF Storage and Retrieval in Jena2*. in *Semantic Web and Databases* 2003. Berlin, Germany.

[7] Kowari, http://www.kowari.org/, (as of 02/24/2005).

[8] Sesame, http://www.openrdf.org/, (as of 02/24/2005).

[9] Simile, *Longwell*, http://simile.mit.edu/longwell/, (as of 02/24/2005).

[10] D. Beckett and J. Grant, *SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes*, http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/, (as of 02/24/2005).

[11] N. Alexander, X. Lopez, S. Ravada, S. Stephens, and J. Wang. *RDF Data Model in Oracle*. in *W3C Workshop on Semantic Web for Life Sciences* 2004. Cambridge, Massachusetts, USA.

[12] Oracle Corporation, *Application Developer's Guide – Object-Relational Features*. 10g Release 1 (10.1) ed. 2003.

[13] Oracle Corporation, *Topology and Network Data Models*. 10g Release 1 (10.1) ed. 2003.

[14] S. Powers, *Practical RDF*. First ed. 2003: O'Reilly & Associates, Inc. 331.

[15] S. Melnik, *Storing RDF in a relational database: Request for Comments*. http://www-db.stanford.edu/~melnik/rdf/db.html, (as of 02/24/2005).

[16] B. McBride, *Jena*. IEEE Internet Computing, 2002 (July/August).

[17] Jena, *Jena2 Database Interface – Database Layout*, http://jena.sourceforge.net/DB/index.html, (as of 02/24/2005).

[18] Jena, *Jena2 Database Interface – Release Notes*, http://jena.sourceforge.net/DB/index.html, (as of 02/24/2005).

[19] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. *Jena: Implementing the Semantic Web Recommendations*. in *World Wide Web Conference* 2004. New York, New York, USA.

[20] D. Beckett, E. Miller, and D. Brickley, *Expressing Simple Dublin Core in RDF/XML*, DCMI Recommendation 2002, http://dublincore.org/ documents/dcmes-xml/, (as of 02/24/2005).

[21] Oracle Corporation, *Oracle XML DB Developer's Guide*, 10g Release 1 (10.1) ed. 2003.

[22] Cytoscape, *Cytoscape 2.0 Manual*, http://www.cytoscape.org/manual/Cytoscape2_0Manual.pdf , (as of 02/24/2005).

[23] cPath, *Cancer Pathway Database – cPath Cytoscape Plugin*, http://www.cbio.mskcc.org/cpath/cytoscape.do, (as of 02/24/2005).

[24] KEGG, *Kyoto Encyclopedia of Genes and Genomes,* http://www.genome.jp/kegg/, (as of 02/24/2005).

[25] W3C, *KEGG RDF Mapping*, http://www.w3.org/2005/02/13-KEGG/, (as of 02/24/2005).

[26] UniProt, *Uniprot: The Universal Protein Resource*, http://www.pir.uniprot.org/, (as of 02/24/2005).